Post-Silicon Customization Using Deep Neural Networks

Kevin Weston[1], Vahid Janfaza[1], Abhishek Taur[1], Dhara Mungra[2]
Arnav Kansal[2], Mohamed Zahran[2], and Abdullah Muzahid[1]

[1] Texas A&M University, TX, USA
[2] New York University, NY, USA

Abstract. Dynamically customizing processor architecture after fabrication, also known as Post-Silicon Customization (PSC) is effective in balancing the conflicting demands of power and performance for various applications. Existing approaches either use application-specific profiles or some adhoc heuristics or simpler machine learning models. These techniques often do not unleash the full potential of PSC as they fail to explore and exploit PSC opportunities to a larger extent. Towards that end, we propose the *first* deep neural network (DNN) based PSC technique, called FORECASTER. FORECASTER exploits several intuitive observations to cope with the long inference latency of a DNN model and boost customization impact. FORECASTER works in two phases. In Phase 1, FORECASTER builds a dataset and then, selects and trains a suitable DNN model offline. In Phase 2, FORECASTER periodically collects hardware telemetry and uses the trained model to customize hardware resources. We provide a detailed design and implementation of FORECASTER and compare its performance against a prior state-of-the-art approach. Our experimental results indicate that on average, FORECASTER provides 2.5X more power efficiency gain over the best static configuration setup while sacrificing less than 1.0% of overall performance and less than 3.5% extra system power. Compared to the prior scheme, FORECASTER increases the power efficiency gain up to 1.5X while reducing the performance degradation by 44%.

Keywords: Deep neural network, FPGA, post-silicon customization.

1 Introduction

Dynamically customizing processor architecture after fabrication, also known as Post-Silicon Customization (PSC) is an effective technique to satisfy the conflicting demands of power and performance for a diverse set of applications [24]. There are three general approaches to implement PSC - profiling, heuristic, and learning-based. Profiling-based approaches profile a particular application on specific hardware or platform and use the profiling information to customize the hardware or software [11, 12, 15]. This line of work requires each program to be instrumented and profiled first. However, the profiled information is useful for only that application. Heuristic-based approaches are built around some heuristics which are often proposed by the architects or programmers based on their experience or intuition [7, 20] on a limited number of hardware or applications. Heuristics often work best for a certain class of applications while other classes

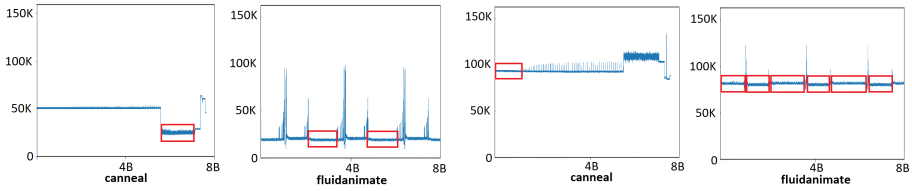
may suffer from poor results. Learning-based approaches try to overcome the limitations of profiling and heuristic-based approaches. At the heart of these approaches are some machine learning models that predict application behavior, processor performance, power consumption, or a combination of these factors. Customization is done based on prediction [8, 22, 24]. Existing learning-based approaches achieve some remarkable results. However, they rely on simple and shallow machine learning models which often fail to unleash the full potential of learning-based approaches. This paper aims to change that by using a deep neural network (DNN) for PSC.

We pinpoint two reasons why existing approaches did not rely on DNN models. *First,* most of the existing works were proposed before the advancement of deep learning techniques [3, 6, 8, 13]. The recent explosion of DNN models and their super-human ability in certain domains, coupled with the availability of hardware accelerators for such models [4, 5] makes DNNs the perfect and timely choice to investigate whether they can improve PSC. *Second,* most of the existing works aim to customize hardware frequently (once in every 100K or less instructions) [22, 24]. Therefore, DNNs with thousands of cycles per inference operation (Table 6) may not be suitable.

To investigate the feasibility of DNNs for PSC, we make two observations. *First*, applications show repetitive execution phases (Section 2). Although phases, when defined at a fine-grained level, might change very frequently, PSC done at such a high frequency does not yield many benefits due to the high customization overhead. Therefore, we have to focus on coarse-grained phases and such phases do not change frequently. *Second, hardware resources that can be customized in the background (without affecting ongoing operations) can boost the customization impact.* Based on these observations, we propose a DNN-based PSC technique, called Forecaster. Forecaster relies on hardware telemetry (a set of hardware performance counters) to approximate how an application behaves and targets four hardware resources for customization - L2 and L3 caches, Branch Target Buffer (BTB), and Prefetcher. We target these structures because they consume significant power [14] and can be customized in the background. Forecaster works in two phases. In Phase 1, it builds a predictive model offline to learn application behavior and the corresponding level of hardware resources to maximize the power efficiency. We use $Instruction\ Per\ Second\ (IPS)^3/Power$ as the metric to calculate power efficiency. This is similar to prior work [8]. Forecaster builds a training dataset based on the data collected from all possible configurations of the selected hardware resources. This dataset is used to train and determine the best DNN model for PSC. In Phase 2, Forecaster initializes a DNN hardware with the model selected from Phase 1. During a program's execution, Forecaster collects hardware telemetry at regular intervals and uses the DNN model to predict the best configuration of hardware resources. Forecaster customizes those resources accordingly to maximize the power efficiency. In summary, we make the following contributions:

1. We propose Forecaster, the *first* PSC technique to use a DNN model. We used a DNN model with over 9 billion parameters.

2. We propose to use a longer customization interval and choose hardware resources carefully to cope up with the long inference delay of a DNN model and boost the customization impact.
3. We provide a detailed design and implementation of FORECASTER using Multi2Sim [25] simulator. Our experimental results using PARSEC 3.0 benchmarks show that on average, FORECASTER provides 2.5X more power efficiency over the best static configuration while sacrificing less than 1.0% of overall performance. Compared to a prior learning scheme [8], FORECASTER increases the power efficiency gain up to 1.5X while reducing the performance degradation by 44%.



(a) branch predictions (b) L1-data accesses

Fig. 1: Number of branch predictions and L1-data accesses over time (x-axis unit is number of instructions). Similarities are highlighted in colored boxes.

2 Intuition

FORECASTER is grounded on two simple hypotheses - *(i) there are significant similarities in execution phases across applications, and (ii) each execution phase requires a specific hardware configuration to maximize the power efficiency without hurting performance.*

To support the first hypothesis, we analyze two applications - *canneal* and *fluidanimate* from Parsec. Figure 1a & 1b show the number of branch predictions and L1-data accesses over execution time. The red-colored boxes in Figure 1a show that one execution phase of *canneal* is similar to two execution phases of *fluidanimate* where the number of branch predictions is steady at around 20K. Thus, the control flow structure of these execution phases between two different applications is similar. If we consider L1-data accesses, Figure 1b shows that one execution phase of *canneal* is similar to 6 other execution phases of *fluidanimate*. Therefore, the data access patterns of these phases of the applications should be similar too. In other words, despite being two completely different applications with different functionalities, *canneal* and *fluidanimate* share a lot of similarities in their execution phases. In addition to this, we notice in both figures that each execution phase usually contains a significant number of instructions (more than a few millions). Therefore, we do not need frequent customization of L1 and BTB structures.

Figure 2 shows the detailed time-series characteristics of L2 and L3 usage as well as branch mispredictions of *fluidanimate*. It shows that the first execution phase (shown in green boxes) has different L2, L3 access, and branch characteristics than the second phase (shown in red boxes). Therefore, the hardware

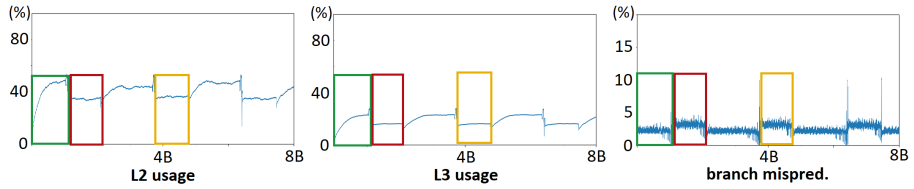


Fig. 2: Time-series data of branch misprediction rate, L2 and L3 usage of *fluidanimate* during execution.

configuration that provides the optimal trade-off between power efficiency and performance for the first phase is different than that of the second phase. For example, since the first phase uses about 45% of L2 and 25% of L3, an optimal cache configuration of the first phase consists of 60% of L2 and 40% of L3. Similarly, the optimal configuration for the second phase is a combination of 40% of L2 and 20% of L3. This clearly demonstrates that every distinct phase requires a different hardware configuration to maximize the power efficiency while maintaining performance. Note that the fourth phase (in yellow box) is quite similar to the second phase and therefore, require the same hardware configuration as the second phase.

3 Background & Related Work

There is a considerable amount of prior work on reconfigurable architecture [2,3, 6–8,11–13,15,20,22,24], which can be grouped into three categories: profiling [11, 12, 15], heuristic [2, 6, 7, 20], and learning-based [3, 8, 13, 22, 24].

Heuristic-Based Techniques: Choi and Yeung [6] perform microarchitectural resources distribution in an SMT processor using hill-climbing algorithm. Petrica et al. [20] present Flicker, a general-purpose multicore architecture that dynamically adapts to varying limits on allocated power. A Flicker core has reconfigurable lanes through the pipeline that allows tailoring an individual core to the running application with lower overhead.

Profiling-Based Techniques: Hubert et al. [11] propose MEMTRACE, a profiling tool that analyzes memory accesses and runtime performance of applications, enabling a variety of optimization opportunities. Ripple [15] introduces a profiling technique that minimizes the instruction cache miss rate. First, the program is profiled offline to get the basic blocks and reconstruct the oracle replacement behavior. Next, Ripple forcefully evicts those basic blocks that is likely to be evicted under the oracle policy by modifying the program binary code.

Learning-Based Techniques: Dubach et al. [8] use Maximum Likelihood Estimation (MLE) to dynamically reconfigure the processor's components. At runtime, whenever the program encounters a new phase, the system enters a profiling period. During this time, The system collects performance counters and converts them into histograms representing the hardware resource usage of that interval. The MLE model uses these histograms as input to predict the optimal configuration to apply for this phase. To reduce noise, the hardware is always reverted to the default configuration during the profiling period, allowing the model to collect unbiased input data. This technique doubles the reconfiguration cost, since the hardware is changed two times per phase: (1) reverting to default configura-

tion for profiling and (2) reconfiguring to the model's prediction. Additionally, the conversion from runtime counters to histograms may produce extra computational latency and hardware support. On the contrary, FORECASTER can work with simple performance counters and does not need a dedicated profiling period, minimizing runtime overheads. Bitirgen et al. [3] combine performance prediction model of multiple applications to get an aggregate performance prediction of the overall resource distribution. Ravi et al. [22] propose CHARSTAR, a clock tree aware resource optimizing mechanism. CHARSTAR incorporates a shallow multi-layer perceptron with one hidden layer to predict the optimal configuration in each execution phase. The model's performance is then tested on single-threaded programs. Tarsa et al. [24] propose a lightweight ML framework that can be distributed through firmware updates to the microcontroller for post-silicon CPUs. The ML model is first trained offline with a collection of applications to avoid statistical blind spots. During execution, the CPU dynamically sets the issue width of a clustered hardware component while clock-gating unused resources.

There is also a well-established line of work that tries to achieve an energy-performance trade-off without any hardware structural adaptation. Prominent works that fall in this category use dynamic voltage-frequency scaling (DVFS) [7, 10, 19]. However, applying this technique in real-world systems can be tricky because reduced frequency means longer execution time.

4 Main Idea: Forecaster

FORECASTER works in two phases - (i) building a model that predicts the best configuration of hardware resources for maximizing the power efficiency (i.e., $IPS^3/Power$) and (ii) changing the hardware resources accordingly. Figure 3 shows the overall workflow. FORECASTER works in the first phase only once using a set of applications whereas the second phase happens at runtime repeatedly during the execution. Both phases use hardware telemetry collected during the execution of an application. The telemetry consists of various hardware event counters that implicitly capture the behavior of the application. The first phase uses telemetry to build a dataset which is used to train a DNN model. The second phase uses the trained DNN model to customize hardware resources.

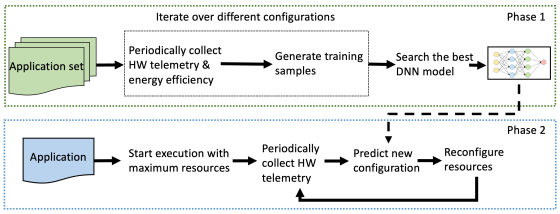


Fig. 3: Overall workflow of FORECASTER.

4.1 Phase 1: Building a Predictive Model

FORECASTER builds a predictive model by first collecting hardware telemetry on a set of benchmarks for different configurations of hardware resources and then, training a DNN model on the dataset.

Tunable Resource	Configuration
BTB Size	0.5K, 1K, 2K, and **4K** Entries
Prefetcher	**On**, Off
L2 (private) cache	256K, 512K, 768K, and **1024K** Bytes
L3 (shared LLC) cache	4M, 8M, 12M, and **16M** Bytes

Table 1: List of reconfigurable hardware. Initial configuration is in bold-face.

Features	Correlation Coefficient
L2 most usage	0.633786
normalized commit float	0.615692
normalized commit mem	0.505695
normalized commit int	0.450328
L1 data access	0.443464
normalized commit ctrl	0.436718
L2 avg eviction rate	0.337137
L2 most hit rate	0.295210
L3 usage	0.264086
branch mispred rate	0.242080

Table 2: Pearson correlation coefficients of counters.

Selecting Hardware Resources As reconfigurable hardware resources, we choose L2 and L3 caches, Branch Target Buffer (BTB) and Prefetcher. We choose caches because they are the most power-hungry resources in a modern chip [14]. We choose the other resources because they can be easily clock-gated without making intrusive changes to the pipeline (Section 4.2). Moreover, as shown in prior work [8], these structures can be customized in the background with minimal impact on performance. Table 1 shows the reconfigurable resources and possible configurations.

Selecting Hardware Telemetry Modern processors provide hundreds of hardware event counters as the telemetry. Not all of them are relevant in deciding how to reconfigure various resources. Therefore, to select the most relevant ones, we use Pearson correlation coefficient. We first extract a large set of 24 microarchitectural counters closely related to those four hardware resources that we want to optimize. These 24 counters capture both program characteristics and their interaction with system components. We then compute the absolute value of Pearson correlation coefficient between the input features and the output label, which is the *power efficiency*. After doing some experiments we decide to select features having the absolute correlation coefficient value greater than 0.20. Table 2 shows the features that have their correlation coefficients greater than the cutoff value. The rest of the features can be discarded to prevent the classifier from learning redundant information. Reducing the size of the feature set minimizes the computational cost and time since we need to train classifiers on large dataset. Those counters, combined with the last interval configuration which are consolidated into 4 inputs, form the final set of 14 input features of our DNN model.

Building Dataset With 4 reconfigurable resources, there are $\mathbb{N} = 4 * 2 * 4 * 4 = 128$ possible configurations. Each application is executed and profiled under each of these configurations. During the execution of an application, FORECASTER collects hardware telemetry, calculates the power efficiency periodically after every $\mathbb{I}$ instructions and records them in a profiling file. Let us call every $\mathbb{I}$ instruction an *Interval*. Let us denote the telemetry as $\mathbb{T} = \{t_i\}_{i=1}^{n}$, where each t_i is an individual hardware counter and the power efficiency as $\mathbb{E}$. Thus, the profiling file contains a set of records of $< \mathbb{T}, \mathbb{E} >$, one record for each interval. FORECASTER keeps the input fixed for an application during profiling. Still, there could be slight perturbation during some execution due to the difference in hardware configurations and thread scheduling (in the case of a multithreaded

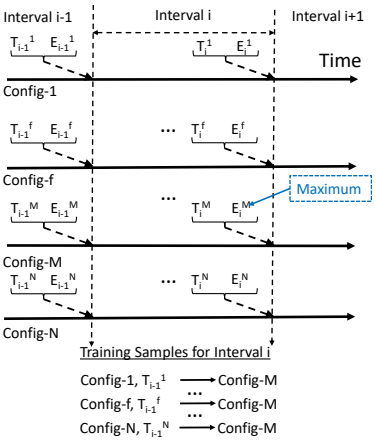


Fig. 4: How training samples are formed from profiles.

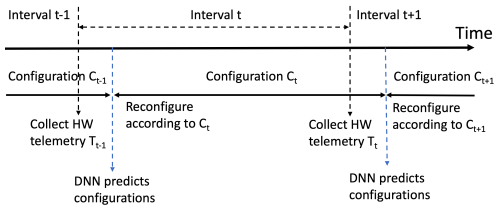


Fig. 5: Timing of various steps of FORECASTER.

application). Therefore, we choose $\mathbb{I}$ to be large enough so that the number of intervals remains the same in every profiling file of an application. As a result, each corresponding interval in different profiling files represents (roughly) the same code region of the application. Whatever little difference that could exist among the code regions of similar intervals, adds noise to the training dataset. Such noise works in favor of DNN models to improve their generality.

Let us consider an interval i. The profiling record for i is $< \mathbb{T}_i^f, \mathbb{E}_i^f >$ in the profiling file for $Config - f$ ($Config - f$ could be any of the $\mathbb{N}$ configurations i.e., $1 \leq f \leq \mathbb{N}$). FORECASTER finds the maximum among $\mathbb{E}_i^1$ to $\mathbb{E}_i^{\mathbb{N}}$. The configuration corresponding to the maximum, say $Config - \mathbb{M}$, provides the highest power efficiency. Therefore, at runtime, when FORECASTER tries to predict the best configuration at the beginning of interval i, it should predict $Config - \mathbb{M}$ as the output of the DNN model. That is why phase 1 forms a training sample by using $\mathbb{T}_{i-1}^f$ as the input and $Config - \mathbb{M}$ as the output. Note that FORECASTER uses $\mathbb{T}_{i-1}^f$ instead of $\mathbb{T}_i^f$ as the input because the telemetry collected at the beginning of interval i is the telemetry corresponding to interval $i-1$. Thus, DNN should be trained to predict $Config - \mathbb{M}$ (the best configuration for interval i) by using the telemetry collected at the beginning of interval i. Figure 4 shows the telemetry and power efficiency of different intervals across different configurations. Last but not least, in addition to $\mathbb{T}_{i-1}^f$, the corresponding configuration i.e., $Config - f$ is provided as part of the input in the training sample. In other words, the training sample is formed by using $< Config - f, \mathbb{T}_{i-1}^f >$ as the input and $Config - \mathbb{M}$ as the output. So, there are $\mathbb{N}$ training samples for interval i.

Selecting a Predictive Model The dataset built previously is used to train a machine learning model. We initially experiment with simple models such as logistic regression or MLE. Our experiments reveal that such simple models are not able to substantially improve the power efficiency of the system (Section 7). We then use more sophisticated models including LSTM, Reinforcement Learning, DNN and see that the fully connected DNN model strikes a good balance between performance and implementation cost. Therefore, we finalize our de-

sign with a fully connected DNN model. To find the best model architecture, FORECASTER searches all possible network configurations (e.g., topology, learning rate, activation functions, etc.) within a constrained search space (e.g., all topologies up to the maximum of H hidden layers and L neurons per layer) and picks the one with the highest accuracy. We use cross-validation for model training and tuning to avoid overfitting. The training strategy for both single-program and multi-program scenarios is discussed in Section 6. The final DNN architecture and its hardware implementation cost is discussed in Section 7.

4.2 Phase 2: Prediction-based Hardware Reconfiguration

During this phase, FORECASTER loads the trained DNN model in a DNN hardware and uses it to predict the configuration of hardware resources for maximizing the power efficiency. When an application starts execution, FORECASTER starts with maximum resources. This prevents any initial slowdown due to insufficient resources. FORECASTER collects hardware telemetry after every interval of $\mathbb{I}$ instructions. Suppose the telemetry after interval t is $\mathbb{T}_t$ and the resource configuration is C_t. FORECASTER uses the DNN hardware with $< C_t, \mathbb{T}_t >$ as the input to infer the new configuration C_{t+1}. Figure 5 shows the timing of the inference step. After DNN hardware calculates the predicted configuration, C_{t+1}, FORECASTER customizes the hardware resources according to C_{t+1}. Now, we describe how each resource is customized.

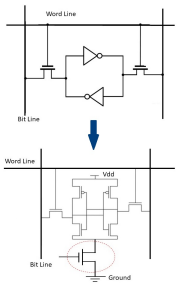
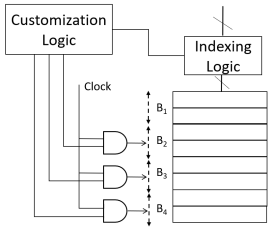
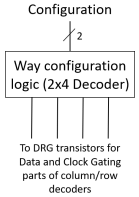


Fig. 6: SRAM Cell Design (6T-MC) with the gated-Vdd shown on the bottom.



(a) BTB.



(b) L2 and L3 caches.

Fig. 7: Logic for customizing hardware components.

L2 and L3 Caches Caches, mainly designed in SRAM (we are not considering eDRAM in this paper) are sources of both static and dynamic power consumption. Dynamic power is consumed in row-decoder, column-decoder, pre-charge circuit and some parts of the core cell and depends on the access pattern. Static power, mainly leakage, is dissipated in every cell of the SRAM cache. With the continuous reduction in transistor sizes and, consequently, the switching threshold voltage of the transistor, static power becomes the major source of power dissipation in caches [26]. Therefore, when we turn off parts of the cache, we want to ensure that we target leakage current. For that, we use gated-ground [1,18,21].

There are several ways of implementing SRAM cells. The one most widely used, due to its relatively high noise immunity, is the 6-transistors Memory Cell

(6T-MC), shown in Figure 6. The left part of the figure shows the gate level of a single SRAM cell. The right part shows the circuit level. There is an extra transistor, shown circled, that is used to reduce leakage current that constitutes the major part of the static power dissipation in caches. In our design of cache resizing, we turn-off individual blocks and never a full-set. Therefore, we can use a single transistor per block. That is, one extra transistor per 64 cells for a 64-byte block. This design does not use more than 4% of extra area with around 5% increase in cache latency [1]. The increased access latency has been taken into consideration in our simulation. When a block is turned-off, that extra transistor is also turned off causing a *stacking effect* that reduces leakage current by orders of magnitude [1].

The next step is to control which blocks will be gated (for static power) and control which parts will be clock-gated to avoid accessing the blocks that are turned-off. From Table 1, we can see that we have four configurations for the cache. We need two bits to represent those configurations. A 2x4 decoder is enough, as shown in Figure 6b. The output of the decoder that is set to one, is used to turn off the corresponding transistors in the data lines. We turn off blocks starting from the last way in each set. For example, in LLC cache, if we want to go from 16MB to 12MB in a 16-way, we turn-off ways 15, 14, 13, and 12. In the current design, FORECASTER only turns off invalid ways, thus incurring no extra data writeback cost. The output of the decoder is also used to clock-gate parts of the column and row decoders to avoid accessing the parts of the cache that are turned off. The customization of the cache does not happen in the critical path of the execution. Therefore, it does not have any effect on performance, except the negligible area and latency increase stated above.

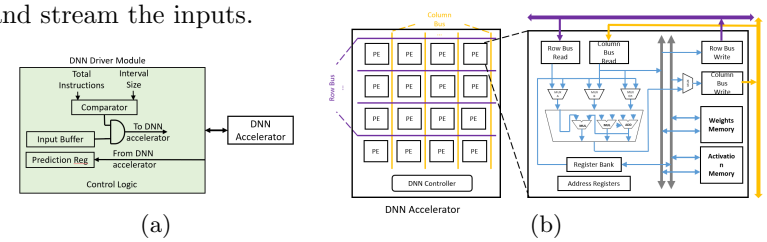
Branch Target Buffer (BTB) BTB has 4 possible configurations (Table 1). Therefore, we can partition BTB into 4 sections - $B1$, $B2$, $B3$, and $B4$ (Figure 6a). For the first configuration (i.e., 0.5K entries), sections ($B2$, $B3$, $B4$) are clock-gated. Similarly, for the second and third configurations, sections ($B3$, $B4$) and ($B4$) are clock-gated respectively. The last configuration does not clock-gate any section at all. On the other hand, Section $B1$ is never clock-gated because at least those entries in BTB are used in all configurations. We add a customization logic that creates the appropriate clock-gating signal to enable the appropriate sections. Moreover, for each configuration, the indexing logic needs to customize the indexing bits accordingly. The extra logic circuits add negligible latency and require less than 200 cycles for customization [8]. The majority of those cycles are hidden in the background. In a multicore processor with one BTB per core, FORECASTER customizes all BTBs to the same configuration. This is done is to simplify the prediction and customization logic in FORECASTER.

Prefetcher Prefetcher is used either completely or not at all. Therefore, the prefetcher is clock-gated entirely or not at all. So, the customization logic simply generates a single clock-gating signal for the entire prefetcher. Customization is completely done in the background.

5 Implementation

In this section, we outline the implementation of DNN in FORECASTER. We implement a simple DNN accelerator in FPGA and add a CPU-side DNN Driver Module to control the operation of the accelerator. The module forms inputs, collects outputs from the accelerator, and sends control signals to the FPGA.

There are many DNN accelerator designs in the literature [5, 9, 23]. We use one similar to the one proposed by Yuanfang Li [17]. Figure 8 shows the overview of our design. The accelerator is constructed as a systolic array of Processing Elements (PEs). The systolic array supports the fast broadcast of inputs and partial sum generation using row and column buses. Each PE contains 2 memories for storing activations and weights, 2 Multiplier, 1 Adder, 2 output buffers for sending results to the row and column buses, 2 input buffers for loading data from the row and column buses, and 4 multiplexers for handling reduction operation during the forward propagation. We consider an extra module for calculating the Softmax function. We distribute the weights of all layers among PEs and stream the inputs.



(a) (b)

Fig. 8: Detailed design of the DNN module.

The DNN driver module has an input buffer, a prediction register, and a control logic. The input buffer is responsible to generate inputs that are provided to the DNN accelerator to infer the predictions. An input consists of various hardware counters and current configuration. Each core collects the counters and current configuration of resources independently and sends them to the driver module after every n (e.g., say n=10,000) instructions. When the module receives counters of at least a total of $\mathbb{I}$ (e.g., $\mathbb{I}$ = interval size) instructions, FORECASTER assumes the start of a new interval. The module aggregates the counters and normalizes each counter with respect to the total instructions of the interval that just finished. The driver module then sends the formed input to the DNN accelerator and receives the predicted configuration. The predicted configuration is stored in the prediction register. The control logic sends the new configuration to the cores and cache controllers to initiate the reconfiguration.

6 Experimental Setup

Interval Size: Determining the right interval size is crucial to strike the optimal balance between performance gain and system overhead. Figure 9 shows the efficiency of different interval sizes across applications, both single and multi-program scenarios. On average, an interval size of 0.5M instructions is the optimal setting, outperforming the second-best by 1.4%.

Simulators and Benchmarks: We use Multi2Sim [25] and McPAT [16] to evaluate FORECASTER and its power consumption. We implement the DNN in the

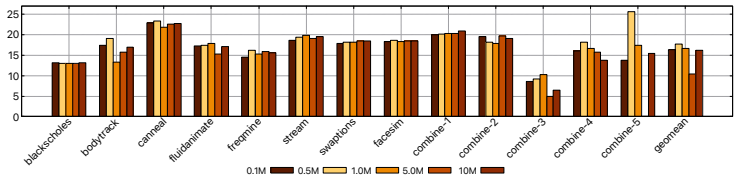


Fig. 9: Efficiency vs. baseline (%) comparison between different interval sizes.

Xilinx FPGA to calculate the latency and overhead. This latency is then used in Multi2Sim. Table 3 shows the hardware parameters for the experiments. We use 8 Parsec 3.0 benchmarks (*blackscholes, bodytrack, canneal, facesim, fluidanimate, freqmine, streamcluster, swaptions*) with small inputs. All benchmarks are run to completion or 1.0B instructions. The interval size $\mathbb{I}$ is set to 0.5M instructions.

Parameter	Value
CPU	8-core @ 2.4Ghz, SMT off
Private L1 cache (I/D)	32KB, 64B line, 8-way
Private L2 Cache	1MB, 64B line, 16-way
Shared L3 Cache	16MB, 64B line, 16-way
Coherence Protocol	Directory-based MOESI

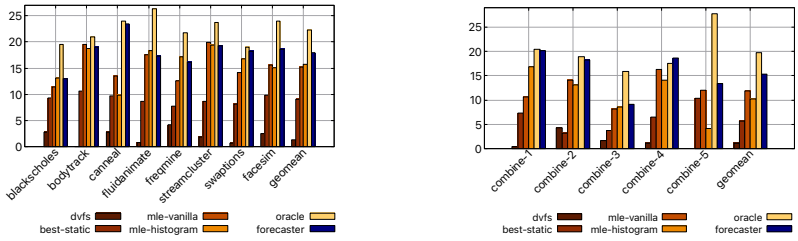
Table 3: Parameters of the simulated hardware.

DNN Training and Tuning: For the single-program scenario, we use leave-one-out cross-validation for model training and tuning. This approach ensures the DNN model is not trained with the application it is optimizing. For the multi-program scenario, we randomly select 5 combinations of programs, each containing 4 different programs. The other 4 that are not chosen are used for training. Two instances of each program are launched during the execution of that combination.

Comparison Work: We compare FORECASTER with 5 other schemes. First, we implement the Maximum Likelihood Estimation (MLE) model from [8]. We call this *MLE-histogram*. Second, we implement a version of FORECASTER using an MLE model (*MLE-vanilla*) instead of a DNN model. This is to compare the performance of the DNN model to the simpler MLE model. Third, to verify the potential of the dynamic optimization scheme, we profile all applications and make two configurations: *best-static* and *oracle*. For *best-static*, we select the best overall static setting for all applications, assuming no dynamic reconfiguration. For the *oracle*, we dynamically apply the optimal setting for all hardware components for each execution phase. This serves as the upper bound for this study. Finally, we also compared our scheme with the DVFS algorithm from [19].

7 Results

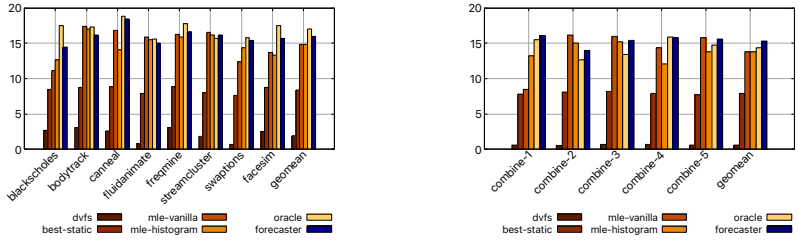
Efficiency Evaluation: The efficiency of single-program and multi-program experiments are shown in Figure 10. In both cases, our scheme outperforms all other tuning techniques, especially in the multi-program scenario. On average, FORECASTER improve the system efficiency by 18.1% compared to the baseline in single-program workloads and 15.8% in multi-program workloads. These improvements account for 80% of the highest achievable efficiency, represented by the oracle configuration.



(a) single-program (b) multi-program

Fig. 10: Normalized efficiency gain vs. baseline (%) of FORECASTER.

Compared to the best static configuration, our technique provides almost 2X more efficiency gain in single-program and 3X more efficiency gain in multi-program scenario. Compared to the MLE-histogram model in [8], FORECASTER provides 15.3% and 49.3% more efficiency in the single and multi-program scenario, respectively. One of the things that separate our work from [8] is that they do not consider the performance of their prediction model in multi-core, multi-program mode, which is a more realistic scenario. The 1.5X performance upgrade in multi-program experiment justifies the use of a more complex DNN model in FORECASTER over the simple MLE technique. The DVFS algorithm we are using puts priority on preserving performance rather than saving power, which is why it has the lowest performance loss, but also the least efficiency gain.



(a) single-program (b) multi-program

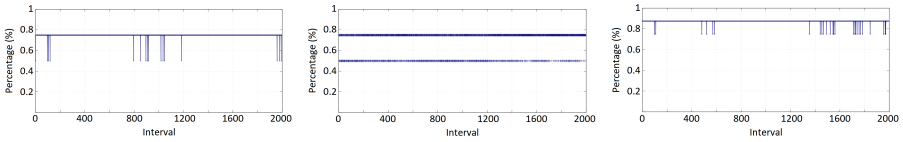
Fig. 11: Normalized power savings vs. baseline (%) of FORECASTER.

This result is achieved thanks to the capability of FORECASTER to accurately predict the hardware demand of applications in each phase to save the most possible amount of power, as shown in Figure 11. In general, FORECASTER manages to save 16% and 15.3% in power compared to the baseline in the single and multi-program scenarios, respectively. For multi-program workload, FORECASTER outperforms all other techniques.

Cache Module	*swaptions*	*combine-5*
L2-0	0.67	0.64
L2-1	0.67	0.62
L2-2	0.67	0.65
L2-3	0.67	0.64
L2-4	0.67	0.62
L2-5	0.67	0.65
L2-6	0.67	0.65
L2-7	0.67	0.65
L3 (last level cache)	0.61	0.67

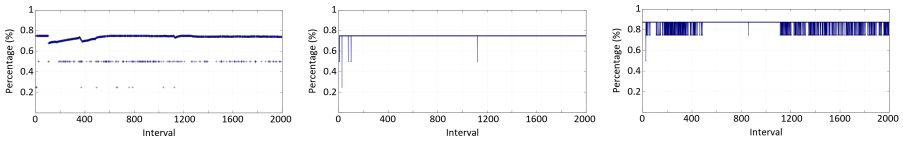
Table 4: Average percentage saving in cache static power of *swaptions* (single-program) and *combine-5* (multi-program).

Detailed Analysis: Figures 12 and 13 show how FORECASTER manages the hardware resources during program execution in single (*swaptions*) and multi-program (*combination-5*) scenarios. FORECASTER accurately estimates the demand of *swaptions*, then turns off excessive resources, saving a lot of power while maintaining the same performance. Sometimes FORECASTER decision cannot be fully satisfied as shown in Figure 13(a). In some intervals, only around 65% to 70% amount of L2 cache is disabled even though the prediction is 75%. This is because those cache blocks are valid. To preserve performance, we do not forcefully turn off resources that are being used. Table 4 shows the break down in cache static power savings of FORECASTER. For the single-program scenario, the amount of power saved is identical between L2 private caches. In multi-program scenario, this number is different because it depends on the application running on the core. In general, using gated-ground technique [1,18,21] to turn off cache blocks, we manage to save approximately 90% of static power of L2 and L3 caches. In *swaptions*, since FORECASTER turns off 75% of L2 and 68% of L3, the actual amounts of static power saved are 67% and 61%, respectively.



(a) L2 Cache (b) L3 Cache (c) BTB

Fig. 12: Average turned off amount of (a) L2, (b) L3, and (c) BTB during the execution of ***swaptions***.



(a) L2 Cache (b) L3 Cache (c) BTB

Fig. 13: Average turned off amount of (a) L2, (b) L3, and (c) BTB during the execution of ***combination-5***.

Runtime Overhead: The runtime cost of the proposed design can be divided into two parts: prediction/reconfiguration latency, and the DNN module power consumption. As for the latency cost, hardware telemetry reading and reconfiguration do not happen in the critical path. The hardware will continue in its old configuration till the decision is made for a new configuration. [24] shows that the reconfiguration time is negligible (tens of cycles). We use this number in our simulation. Overall, our approach manages to reduce the IPS degradation by about 44% compared to MLE-histogram, as shown in Table 5.

The main power cost of FORECASTER comes from the DNN driver module and the Processing Elements (PEs). With 16x16 configuration, the PE array of FORECASTER consumes a total power of 4.94W, as shown in Table 6. For the DNN driver module, its total power consumption measured by McPAT is only 0.032W. Altogether, the total power usage of FORECASTER is 4.97W. As the system overall power consumption measured by McPAT is 142W for the

Technique	single-program	multi-program	mean
DVFS	(0.1)	0.2	0.1
Best-static	0.0	(0.8)	(0.4)
FORECASTER	(0.3)	(0.7)	(0.5)
MLE-histogram	(0.4)	(1.4)	(0.9)
MLE-vanilla	(0.6)	(1.3)	(1.0)
Oracle	0.5	0.9	0.7

Table 5: IPS degradation (%) vs. baseline. Negative numbers in parentheses.

PE Array	Frequency (MHz)	Latency	Slice Reg	Power (W)	
				Static	Dynamic
8*8	268	6352	79566	0.20	2.51
12*12	258	3200	93179	0.21	4.36
16*16	**247**	**1896**	**109972**	**0.22**	**4.72**

Table 6: Cost of different DNN hardware. This power usage is less than 3.5% of the overall system power.

single-program scenario and 153W for the multi-program scenario, the power consumption of FORECASTER is just 3.49% and 3.24% extra. Furthermore, since the DNN model is only used once per 0.5M instructions, its actual energy cost is minimal.

Hardware Implementation Cost The hardware cost consists of the DNN hardware and the extra hardware used to implement the knobs. The DNN uses a four-hidden-layer fully connected neural network with the neuron configuration of *384/384/256/256*. There are also an input layer of 14 neurons and an output layer of 128 neurons. We use *ReLU* activation for the input and hidden layers and *Softmax* for the output layer. For the hardware implementation, we consider several design points as shown Table 6. We use 16*16 PE array size in our final design.

The hardware needed for the knobs is straightforward. The prefetcher is just clock-gated as the knob is on/off. The BTB also uses clock-gating depending on the configuration. We have four configurations so a small 2x4 decoder will do the job as shown in the customization logic in Figure 6a. Clock gating the cache ways is simplified by the fact that the way-reconfiguration logic, shown in Figure 6b, never gates a valid entry so no change to the cache controller or coherence hardware is needed. The way-reconfiguration logic is not complicated because it exploits the fact that large caches (such as L3) are usually partitioned. Therefore we have one logic circuitry per partition.

8 Conclusions

The work presents the *first* DNN-based PSC technique, called FORECASTER. FORECASTER exploits two intuitive observations to cope with the long inference latency of a DNN model and boost customization impact. FORECASTER works in two phases - offline training and online reconfiguration. We provide a detailed design and implementation of FORECASTER and compare its performance against a prior state-of-the-art approach. Overall, FORECASTER provides 2.5X and 1.5X more power efficiency gain over the best static configuration and prior state-of-the-art approach.

Acknowlegement

We thanks the reviewers and the members of PALab research group for valuable feeback. This work is supported by Texas A&M University Faculty Startup Grant, and NSF Grant No. 1931078.

References

1. A. Agarwal, Hai Li, and K. Roy, "Drg-cache: a data retention gated-ground cache for low power," in *Design Automation Conference*, 2002.
2. R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *MICRO*, 2000.
3. R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *MICRO*, 2008.
4. T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.
5. Y. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, 2016.
6. S. Choi and D. Yeung, "Learning-based smt processor resource distribution via hill-climbing," in *ISCA*, 2006.
7. Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "Coscale: Coordinating cpu and memory system dvfs in server systems," in *MICRO*, 2012.
8. C. Dubach, T. M. Jones, E. V. Bonilla, and M. F. P. O'Boyle, "A predictive model for dynamic microarchitectural adaptivity control," in *MICRO*, 2010.
9. H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.
10. J. Haj-Yahya, M. Alser, J. Kim, A. G. Yağlıkçı, N. Vijaykumar, E. Rotem, and O. Mutlu, "Sysscale: Exploiting multi-domain dynamic voltage and frequency scaling for energy efficient mobile processors," in *ISCA*, 2020.
11. H. Hubert and B. Stabernack, "Profiling-based hardware/software co-exploration for the design of video coding architectures," *IEEE TCSVT*, vol. 19, 2009.
12. Intel, "Profile-Guided Optimization (PGO)," https://software.intel.com/content/www/us/en/develop-/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/profile-guided-optimization-pgo.html.
13. E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *ISCA*, 2008.
14. C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *MICRO*, 2006.
15. T. A. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "Ripple: Profile-guided instruction cache replacement for data center applications," in *ISCA*, 2021.
16. S. Li, H. Ann, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.
17. Y. Li and A. Pedram, "Caterpillar: Coarse grain reconfigurable architecture for accelerating the training of deep neural networks," in *2017 IEEE 28th International Conference on ASAP*, 2017.
18. A. Manan, "Efficient 16 nm sram design for fpga's," in *SPIN*, 2018.
19. V. Pallipadi and A. Starikovskiy, "The ondemand governor," 2006.
20. P. Petrica, A. M. Izraelevitz, D. H. Albonesi, and C. A. Shoemaker, "Flicker: A dynamically adaptive architecture for power limited multicore systems," in *ISCA*, 2013.
21. M. Powell, Se-Hyun Yang, B. Falsafi, K. Roy, and N. Vijaykumar, "Reducing leakage in a high-performance deep-submicron instruction cache," *VLSI*, 2001.
22. G. S. Ravi and M. H. Lipasti, "Charstar: Clock hierarchy aware resource scaling in tiled architectures," in *ISCA*, 2017.
23. B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ISCA*, 2016.
24. S. J. Tarsa, R. B. R. Chowdhury, J. Sebot, G. Chinya, J. Gaur, K. Sankaranarayanan, C.-K. Lin, R. Chappell, R. Singhal, and H. Wang, "Post-silicon cpu adaptation made practical using machine learning," in *ISCA 2019*.
25. R. Ubal, J. Sahuquilo, S. Petit, and P. López, "Multi2sim: A simulation framework to evaluate multicore-multithreaded processors," in *SBAC-PAD*, 2007.
26. A. Wiltgen, K. A. Escobar, A. I. Reis, and R. P. Ribas, "Power consumption analysis in static cmos gates," in *SBCCI*, 2013.